Int J Softw Tools Technol Transfer (2016) 18:169–185
DOI 10.1007/s10009-015-0379-9

TACAS 2014

Many-core on-the-fly model checking of safety properties using GPUs

Anton Wijs[1,2] · **Dragan Bošnački**[1]

Published online: 29 April 2015

Abstract Model checking is an automatic method to formally verify the correctness of a system specification. Such model checking specifications can be viewed as implicit descriptions of a large directed graph or state space, which, for most model checking operations, needs to be analysed. However, construction or on-the-fly exploration of the state space is computationally intensive and often can be prohibitive in practical applications. In this work, we present techniques to perform graph generation and exploration using general purpose graphics processors (GPUs). GPUs have been successfully applied in multiple application domains to drastically speed up computations. We explain the limitations involved when trying to achieve efficient state space exploration with GPUs and present solutions how to overcome these. We discuss the possible approaches involving related work and propose an alternative, using a new hash table approach for GPUs. As input, we consider models that can be represented by a fixed number of communicating finite-state Labelled Transition Systems. This means that we assume that all variables used in a model range over finite data domains. Additionally, we show how our exploration technique can be extended to detect deadlocks and check safety properties on-the-fly. Experimental evaluations with our prototype implementations show significant speed-ups compared to the established sequential counterparts.

Keywords GPU · Model checking · Safety properties · Graph search · Refinement

This work was sponsored by the NWO Exacte Wetenschappen, EW (NWO Physical Sciences Division) for the use of supercomputer facilities, with financial support from the Nederlandse Organisatie voor Wetenschappelijk Onderzoek (Netherlands Organisation for Scientific Research, NWO).

✉ Anton Wijs
A.J.Wijs@tue.nl

Dragan Bošnački
D.Bosnacki@tue.nl

1 Eindhoven University of Technology, Eindhoven, The Netherlands

2 RWTH Aachen University, Aachen, Germany

1 Introduction

Model checking [3] is an automatic technique used to formally verify properties of specifications of complex, safety-critical (usually embedded) systems. The technique often involves many time- and memory-demanding computations. The analysis of the specification results in building a graph, or state space, that describes the complete potential behaviour of the system. Many model checking algorithms rely on *on-the-fly state space exploration*, meaning that the state space size is not known in advance, but it is rather generated on demand by interpreting the specification.

In recent years, general-purpose graphics processing units (GPUs) have been used in many research areas, beyond the realm of computer graphics, to accelerate computations. Also, in model checking GPUs have been successfully applied to perform computations when the state space is given a priori. Examples of such applications are verifying Markov Chains w.r.t. probabilistic properties [9,10,44], decomposing state spaces into strongly connected components [5,46], comparing and minimising state spaces using some notion of bisimilarity [43], and checking state spaces w.r.t. LTL properties [4,6].

However, very few attempts have been made to perform the exploration itself entirely using GPUs, due to it not naturally fitting the data parallel approach of GPUs. In [45], we were the first to propose a way to do so, and since then, Bartocci et al. [7] have proposed an approach to use GPUs with the SPIN model checker. Even though current GPUs have a limited amount of memory, we believe it is relevant to investigate the possibilities of GPU state space exploration, if only to be prepared for future hardware developments (for example, GPUs are already being integrated in CPUs). We also believe that the gained insights can be relevant for solving other on-the-fly graph problems.

In this article, we first describe several options to implement basic state space exploration, i.e. reachability analysis, for explicit-state model checking on GPUs. To keep the input finite and straightforward, we consider models that can be represented by a finite number of processes, where each process can be modelled as a finite-state labelled transition system (LTS). This means that we practically restrict models to using variables that only take values from finite data domains. We focus on CUDA-enabled GPUs of NVIDIA, but the options can also be implemented using other interfaces. Next, we discuss how the proposed technique can be extended to on-the-fly check for deadlocks and violations of safety properties.

We experimentally compare the different implementation options using various GPU configurations. Based on the obtained results, we draw conclusions.

Where relevant, we use techniques from related work, but practically all related implementations are focussed on explicit graph searching, in which the explicit graph is given, as opposed to on-the-fly constructing the graph.

This article is based on [45] and extends it in various ways: First of all, the many-core exploration technique is discussed in much more detail. Second, checking for deadlocks and violations of safety properties is a new addition, and third, additional experimental results obtained using the Cuckoo hash table by Alcantara et al. [2] are reported.

The structure of the article is as follows: in Sect. 2, the required background information is given. Then, Sect. 3 contains the description of several implementations of many-core state space exploration using different extensions. How to extend those implementations to support on-the-fly deadlock detection and checking for violations of safety properties is discussed in Sect. 4. In Sect. 5, experimental results are given, and finally, Sect. 6 contains conclusions and discusses possible future work.

2 Background and related work

2.1 State space exploration

The first question is how a specification should be represented. Most descriptions, unfortunately, are not very suitable for our purpose, since they require the dynamic construction of a database of data terms during the exploration. GPUs are particularly unsuitable for dynamic memory allocation. We choose to use a slightly modified version of the *networks of LTSs* model [25]. In such a network, the possible behaviour of each process or component of the concurrent system design is represented by a *process* LTS, or *Labelled Transition System*. An LTS can be defined as follows:

Definition 1 (*Labelled transition system*) A *Labelled Transition System* $\mathcal{G}$ is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, s^0 \rangle$, where $\mathcal{S}$ is a (finite) set of states, $\mathcal{A}$ is a set of actions or transition labels, $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is a transition relation, and $s^0 \in \mathcal{S}$ is the initial state.

Actions in $\mathcal{A}$ are denoted by a, b, c, etc. We use $s_0 \xrightarrow{a} s_1$ to indicate that $\langle s_0, a, s_1 \rangle \in \mathcal{T}$. If $s_0 \xrightarrow{a} s_1$, an action a can be performed in state s_0, and doing so leads to state s_1.

A process LTS describes all potential behaviour of a corresponding process. Each transition has a label indicating the event that is fired by the process.

A network of LTSs defines a concurrent system by means of a finite list of process LTSs, and a synchronisation mechanism, describing how those LTSs may interact. Thus, such a network is able to capture the semantics of specifications with *finite-state* processes at a level where all data have been abstracted away and only states remain. It is used in particular in the CADP verification toolbox [15]. In this paper, we restrict synchronisation mechanisms to *multi-rendezvous* as used in LOTOS [21]: we allow rules saying that if a particular number of LTSs can perform a particular action a, then they can synchronise on that action, resulting in the system as a whole performing an a action.

Both infinite-state processes and more general synchronisation mechanisms are out of the scope. Concerning the latter, we experienced that many systems can be described using multi-rendezvous rules. Nevertheless, removing both limitations is considered future work.

We define the notion of a network of LTSs (with multi-rendezvous synchronisation) as follows:

Definition 2 (*Network of LTSs*) A *network of LTSs* $\mathcal{M}$ of size n is a tuple $\langle \Pi, \mathcal{V} \rangle$, where

- Π is a vector of n (process) LTSs. For each $i \in 1..n$, we write $\Pi[i] = \langle \mathcal{S}_i, \mathcal{A}_i, \mathcal{T}_i, s_i^0 \rangle$, and $s_1 \xrightarrow{a}_i s_2$ is shorthand for $\langle s_1, a, s_2 \rangle \in \mathcal{T}_i$;
- $\mathcal{V}$ is a finite set of *synchronisation rules*. A synchronisation rule is a tuple $\langle \bar{t}, a \rangle$, where a is an action label, and $\bar{t}$ is a vector of size n called a *synchronisation vector*, in which for all $i \in 1..n$, $\bar{t}[i] \in \{a, \bullet\}$, where $\bullet$ is a special symbol denoting that $\Pi[i]$ performs no action.

Besides a finite number of LTSs, a network also contains a finite set $\mathcal{V}$ of synchronisation rules, describing how behaviour of different processes should synchronise. Through this mechanism, it is possible to model synchronous communication between processes. Each rule $\langle \bar{t}, a\rangle$ consists of an action a on which it is applicable, and a vector $\bar{t}$ of size n, describing which process LTSs need to be involved to have successful synchronisation. The result of a synchronisation is again a. As an example, consider the two LTSs at the top in Fig. 1, together defining a network with $n = 2$ of a simple traffic light system specification, where process $\Pi[1]$ represents the behaviour of a traffic light (the states representing the colours of the light) and process $\Pi[2]$ represents a pedestrian. We also have $\mathcal{V} = \{(\langle \mathit{crossing}, \mathit{crossing}\rangle, \mathit{crossing})\}$, meaning that there is only a single synchronisation rule, expressing that the *crossing* event of $\Pi[1]$ can only be fired if event *crossing* of $\Pi[2]$ is fired at the same time, resulting in the event *crossing* being fired by the system as a whole.

A network of LTSs $\mathcal{M} = \langle \Pi, \mathcal{V}\rangle$ is an implicit description of all possible system behaviour. We call the explicit behaviour description of a network $\mathcal{M}$ the *system LTS LTS*$(\mathcal{M})$. It can be obtained by combining the $\Pi[i]$ according to the rules in $\mathcal{V}$. Here, we define the system LTS of a network differently from Lang [25]. Namely, we say that an *independent* action, i.e. an action that requires no synchronisation with other actions, does not require a matching synchronisation rule to be fireable. A consequence of this is that we do not allow nondeterministic synchronisation of actions. However, in our experience, this only rarely poses a real limitation in practice, while it allows for more compact definitions of synchronisation mechanisms. In the following definition, the set $\mathcal{A}_i^{ind}$ refers to the set of all actions of $\Pi[i]$ that do not require synchronisation, i.e. $\mathcal{A}_i^{ind} = \{a \in \mathcal{A}_i \mid \neg\exists\langle \bar{t}, a\rangle \in \mathcal{V}.\bar{t}[i] = a\}$.

Definition 3 Given a network of LTSs $\mathcal{M} = \langle \Pi, \mathcal{V}\rangle$. The LTS describing all potential behaviour of $\mathcal{M}$ explicitly is called its *system LTS LTS*$(\mathcal{M}) = \langle \mathcal{S}_\mathcal{M}, \mathcal{A}_\mathcal{M}, \mathcal{T}_\mathcal{M}, s^0_\mathcal{M}\rangle$, with

- $s^0_\mathcal{M} = \langle s^0_1, \ldots, s^0_n\rangle$;
- $\mathcal{A}_\mathcal{M} = \bigcup_{i\in 1..n} \mathcal{A}_i$;
- $\mathcal{S}_\mathcal{M} = \mathcal{S}_1 \times \cdots \times \mathcal{S}_n$;
- $\mathcal{T}_\mathcal{M}$ is the smallest transition relation satisfying

  1. $\forall\langle \bar{t}, a\rangle \in \mathcal{V}, i \in 1..n.$

  $$\left(\begin{array}{c}(\bar{t}[i] = \bullet \wedge \bar{s}'[i] = \bar{s}[i])\\ \vee\ (\bar{t}[i] = a \wedge \bar{s}[i] \xrightarrow{a}_i \bar{s}'[i])\end{array}\right) \implies \bar{s} \xrightarrow{a} \bar{s}'$$

  2. $\forall i \in 1..n, a \in \mathcal{A}_i^{ind}.$

  $$\bar{s}[i] \xrightarrow{a}_i \bar{s}'[i] \implies \bar{s} \xrightarrow{a} \bar{s}',$$

  with $\forall j \in 1..n\backslash\{i\}.\bar{s}'[j] = \bar{s}[j]$.

When doing state space exploration, we are actually interested in the subgraph of $LTS(\mathcal{M})$ that is reachable from $s^0_\mathcal{M}$.

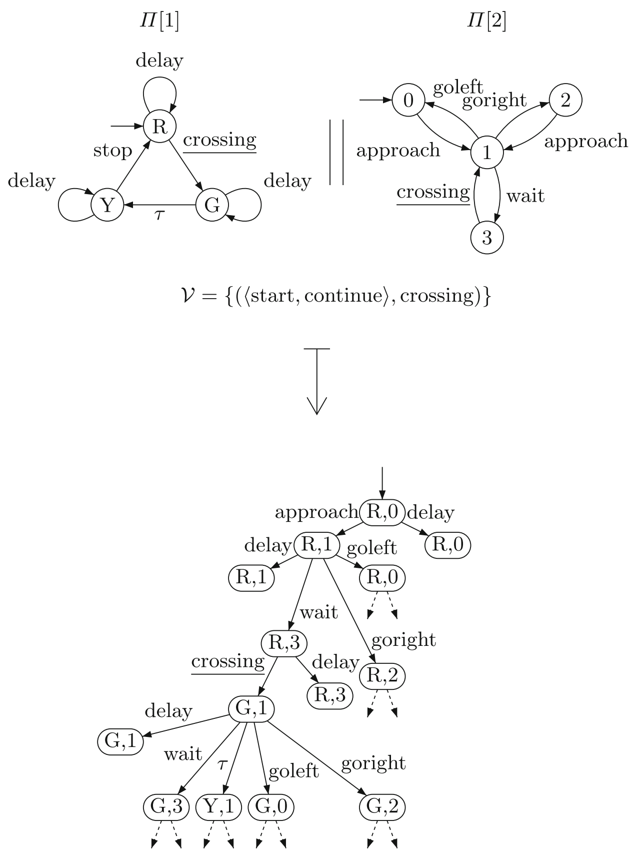

Fig. 1 Exploring the state space of a traffic light specification

This can be discovered on-the-fly as follows: First, the initial states of the process LTSs (in Fig. 1 these are indicated by an incoming transition without a source state) are combined into a system state vector $s^0_\mathcal{M}$. For the traffic light system, we have $s^0_\mathcal{M} = \langle R, 0\rangle$. In general, given a vector $\bar{s}$, the corresponding state of $\Pi[i]$, with $i \in 1..n$, is $\bar{s}[i]$. In the following, we refer with $\mathcal{V}_i^a$ to the synchronisation vectors in $\mathcal{M}$ that are applicable on action a of $\Pi[i]$. It is defined as

$$\mathcal{V}_i^a = \{\bar{t} \mid \langle \bar{t}, a\rangle \in \mathcal{V} \wedge \bar{t}[i] = a\}$$

The set of outgoing transitions of $\bar{s}$ (and their corresponding target states or successors of $\bar{s}$) involving a particular $\Pi[i]$ can now be determined using the following check for each transition $\bar{s}[i] \xrightarrow{a} p_i$, with p_i a state of $\Pi[i]$:

1. if $\mathcal{V}_i^a = \emptyset$, then $\bar{s} \xrightarrow{a} \bar{s}'$ with $\bar{s}'[i] = p_i \wedge \forall j \in 1..n\backslash\{i\}. \bar{s}'[j] = \bar{s}[j]$
2. else, for all $\bar{t} \in \mathcal{V}_i^a$, check the following: if we have for all $j \in 1..n\backslash\{i\}$ that $\bar{t}[j] = \bullet \vee \bar{s}[j] \xrightarrow{a} p_j$, then $\bar{s} \xrightarrow{a} \bar{s}'$ with $\forall j \in 1..n.(\bar{t}[j] = \bullet \wedge \bar{s}'[j] = \bar{s}[j]) \vee (\bar{t}[j] \neq \bullet \wedge \bar{s}'[j] = p_j)$

The first case is applicable to all *independent* transitions, i.e. transitions on which no rule is applicable; hence they can be fired individually, and, therefore, directly 'lifted' to the system level. The second case involves applying synchronisation rules. If we perform the above check for all $\Pi[i]$ on each visited state vector, starting with $s^0_{\mathcal{M}}$, then the reachable system state space will be fully explored. In Fig. 1, part of the system state space obtained by applying the defined checks on the traffic network is displayed at the bottom.

2.2 GPU programming

NVIDIA GPUs can be programmed using the CUDA interface, which extends the C and FORTRAN programming languages. These GPUs contain tens of streaming multiprocessors (SM) (see Fig. 2, with N the number of SMs), each containing a fixed number of streaming processors (SP), e.g. 192 for the Kepler K20m GPU, and fast on-chip *shared memory*. Each SM employs single instruction, multiple data (SIMD) techniques, allowing for data parallelisation. A single instruction stream is performed by a fixed size group of threads called a *warp*. Threads in a warp share a program counter and hence perform instructions in lock-step. Due to this, *branch divergence* can occur within a warp, which should be avoided: for instance, consider the if-then-else construct **if (C) then A else B**. If a warp needs to execute this, and for at least one thread **C** holds, then *all* threads must step through **A**. It is, therefore, possible that the threads must step together through both **A** and **B**, thereby decreasing performance. The size of a warp is fixed and depends on the GPU type; usually it is 32 and we refer to it as *WarpSize*. A *block* of threads is a larger group assigned to a single SM. The threads in a block can use the shared memory to communicate with each other. An SM, however, can handle many blocks in parallel. Instructions to be performed by GPU threads can be defined in a function called a *kernel*.

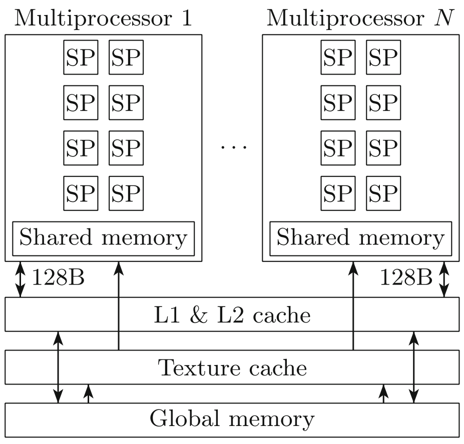

Fig. 2 Hardware model of CUDA GPUs

When launching a kernel, one can specify how many thread blocks should execute it, and how many threads each block contains (usually a power of two). We refer to the number of launched blocks with *NrOfBlocks* and to the size of a block with *BlockSize*. Each SM then schedules all the threads of its assigned blocks up to the warp level. Data parallelisation can be achieved using the predefined keywords *BlockId* and *ThreadId*, referring to the ID of the block a thread resides in, and the ID of a thread within its block, respectively.

From the predefined keywords *NrOfBlocks*, *BlockSize*, *BlockId*, and *ThreadId*, it is possible to derive other useful keywords. For instance, we refer with *WarpId* to the block-local ID of a warp, which can be defined as $\mathit{WarpId} = \mathit{ThreadId}/\mathit{WarpSize}$. Furthermore, we define $\mathit{WarpTId} = \mathit{ThreadId} \bmod \mathit{WarpSize}$ as the ID of a thread within its warp, and $\mathit{GlobalWarpId} = (\mathit{BlockSize}/\mathit{WarpSize}) \cdot \mathit{BlockId} + \mathit{WarpId}$ refers to the *global* ID of a warp; it can be determined by computing how many warps are run per block, multiplying this with the block ID of the thread (thereby obtaining the number of warps with an ID below the one of the thread), and finally adding the block-local warp ID. Finally, the number of warps *NrOfWarps* can be determined by computing $\mathit{NrOfWarps} = (\mathit{BlockSize}/\mathit{WarpSize}) \cdot \mathit{NrOfBlocks}$.

Most of the data used by a GPU application reside in *global memory* or device memory. It embodies the interface between the host (CPU) and the kernel (GPU). Depending on the GPU type, its size is currently between 1 and 12 GB. It has a high bandwidth, but also a high latency; therefore, memory caches are used. The cache line of most current NVIDIA GPU L1 and L2 caches is 128 Bytes, which directly corresponds to each thread in a warp fetching a 32-bit integer. If memory accesses in a kernel can be coalesced within each warp, efficient fetching can be achieved. In this case, the threads in a warp perform a single fetch together filling one cache line. When memory accesses are not coalesced, the accesses of threads in a warp are performed using different fetches that are serialised by the GPU, thereby losing many clock-cycles. This plays an important role in the hash table implementation we propose.

Finally, read-only data structures in global memory can be declared as *textures*, by which they are connected to a *texture cache*. This may be beneficial if access to the data structure is expected to be random, since the cache may help in avoiding some global memory accesses.

2.3 Sparse graph search on GPUs

In general, the most suitable search strategy for parallelisation is breadth-first search (BFS) since each search level is a set of vertices that can be distributed over multiple workers. Two operations dominate in BFS: *neighbour gathering*, i.e. obtaining the list of vertices reachable from a given vertex via one edge, and *status lookup*, i.e. determining whether a vertex has already been visited before. There exist many parallelisations of BFS; here, we will focus on GPU versions.

Concerning model checking, Edelkamp and Sulewski [14] propose a GPU on-the-fly exploration approach using both the CPU and GPU, restricting the GPU to neighbour gathering. It uses bitstate hashing; hence it is not guaranteed to be exhaustive. Barnat et al. [4,6] check explicitly known state spaces w.r.t. LTL properties.

The vast majority of GPU BFS implementations are quadratic parallelisations, e.g. [12,18]. To mitigate the dependency of memory accesses on the graph structure, each vertex is considered in each iteration, yielding a complexity of $\mathcal{O}(|V|^2 + |E|)$, with V the set of vertices and E the set of edges. Hong et al. [20] use entire warps to obtain the neighbours of a vertex.

There are only a few linear parallelisations in the literature: Luo et al. [26] describe a hierarchical scheme using serial neighbour gathering and multiple queues to avoid high contention on a single queue. Merrill et al. [29] suggest an approach using prefix sum and perform a thorough analysis to determine how gatherings and lookups need to be placed in kernels for maximum performance.

All these approaches are, however, not directly suitable for on-the-fly exploration. First of all, they implement status lookups by maintaining an array, but in on-the-fly exploration, the required size of such an array is not known a priori. Second of all, they focus on using an adjacency matrix, but for on-the-fly exploration, this is not available, and the memory access patterns are likely to be very different.

Related to the first objection, the use of a hash table seems unavoidable. Not many GPU hash table implementations have been reported, but the ones by Alcantara et al. [1,2] are notable. They are both based on *Cuckoo hashing* [33]. In Cuckoo hashing, collisions are resolved by shuffling the elements along to new locations using multiple hash functions. Whenever an element must be inserted, and hash function h_0 refers it to a location l already populated by another element, then the latter element is replaced using the next hash function for that element, i.e. if it was placed in l using hash function h_i, then function $h_{i+1} \mod H$, with H the number of hash functions, is used. Alcantara et al. [2] suggest to set $H = 4$.

Also, Alcantara et al. [1,2] perform a comparison to radix sorting, in particular to the implementation of Merrill and Grimshaw [30]. On a GPU, sorting can achieve high throughput, due to the regular access patterns, making list insertion and sorting faster than hash table insertion. Lookups, however, are slower than hash table lookups if one uses binary searches, as is done by Alcantara et al. [1,2]. An alternative is to use B-trees for storing elements, improving memory access patterns by grouping the elements in warp-segments.[1]

[1] See http://www.moderngpu.com (visited 18/4/2013).

Algorithm 1 State space exploration

```
Require: network $\langle \Pi, \mathcal{V} \rangle$, initial state $\bar{s}_I$
     $Open, Visited \leftarrow \{\bar{s}_I\}$
2:   while $Open \neq \emptyset$ do
         $\bar{s} \leftarrow Open$; $Open \leftarrow Open \setminus \bar{s}$
4:       for all $\bar{s}' \in$ constructSystemSuccs($\bar{s}$) do
             if $\bar{s}' \notin Visited$ then
6:               $Visited \leftarrow Visited \cup \{\bar{s}'\}$
                 $Open \leftarrow Open \cup \{\bar{s}'\}$
```

Although we have chosen to use a hash table approach (for on-the-fly exploration, we experience that the sorting approach is overly complicated, requiring many additional steps), we will use this idea of warp-segments for our hash table.

Finally, more recently, GPUs have been employed to speed up on-the-fly state space exploration and safety property checking of the SPIN model checker [7]. The proposed technique is based on the multi-core BFS algorithm of SPIN [19] and uses the GPU Cuckoo hashing technique of Alcantara et al. However, by doing so, they restrict themselves to state vectors of at most 64 bits since the Cuckoo hashing technique depends on atomic element insertion. Because of this restriction, we have chosen to develop an alternative hashing technique.

3 GPU parallelisation

Algorithm 1 provides a high-level view of state space exploration. As in BFS, one can clearly identify the two main operations, namely *successor generation* (line 4), analogous to neighbour gathering, and *duplicate detection* (line 5), analogous to status lookup. Finally, at lines 6–7, states are added to the work sets, *Visited* being the set of visited states and *Open* being the set of states yet to be explored (usually implemented as a queue). In the next subsections, we will discuss our approach to implementing these operations.

3.1 Data encoding

As mentioned before, memory access patterns are usually the main cause for performance loss in GPU graph traversal. The first step to minimise this effect is to choose appropriate encodings of the data. Figure 3 presents in the top left corner how we encode a network into three 32-bit integer arrays. The first, called *ProcOffsets*, contains the start offset for each of the $\Pi[i]$ in the second array. The second array, *StateOffsets*, contains the offsets for the source states in the third array. Finally, the third array, *TransArray*, actually contains encodings of the outgoing transitions of each state. As an example, let us say we are interested in the outgoing transitions of state 5 of process LTS $\Pi[8]$, in some given network. First, we look at position 8 in *ProcOffsets*, and find that the states of that

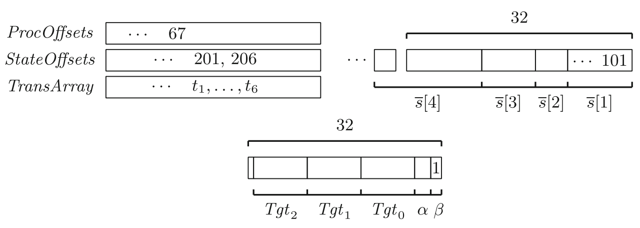

Fig. 3 Encodings of a network, a state vector and a transition

process are listed starting from position 67. Then, we look at position 67 + 5 in *StateOffsets*, and we find that the outgoing transitions of state 5 are listed starting at position 201 in *TransArray*. Moreover, at position 67 + 6 in *StateOffsets*, we find the end of that list. Using these positions, we can iterate over the outgoing transitions in *TransArray*.

One can imagine that these structures are practically going to be accessed randomly when exploring. However, since these data are never updated, we can store the arrays as textures, thereby using the texture caches to improve access.

Besides this, we must also encode the transition entries themselves. This is shown at the bottom of Fig. 3. Each entry fills a 32-bit integer as much as possible. It contains the following information: the lowest bit (β) indicates whether or not the transition is independent. The next $\log_2(|\mathcal{A}_{\mathcal{M}}|)$ number of bits encodes the transition label (α). Note that $\mathcal{A}_{\mathcal{M}}$ can be constructed a priori as the union of the $\mathcal{A}_i$. We encode the labels, which are basically strings, by integer values, sorting the labels occurring in a network alphabetically. After that, given that the transition belongs to $\Pi[i]$, each $\log_2(|\mathcal{S}_i|)$ bits encode a target state. If there is non-determinism w.r.t. the involved label from the source state, multiple target states will be listed, possibly continuing in subsequent transition entries.

In the top right corner of Fig. 3, the encoding of state vectors is shown. These are simply concatenations of encodings of process LTS states. Depending on the number of bits needed per LTS state, which in turn depends on the number of states in the LTSs, a fixed number of 32-bit integers is required per vector.

Finally, the synchronisation rules need to be encoded. Multi-rendezvous rules can be encoded as bit sequences of size n, where for each process LTS, 1 indicates that the process should participate, and 0 that it should not participate in synchronisation. Two integer arrays then suffice, one containing these encodings, the other containing the offsets for all the labels. In that way, when a state vector $\bar{s}$ must be explored, one can fetch for each label on at least one outgoing transition of an $\bar{s}[i]$ all bit sequences relevant for that label, and check for each sequence whether the condition encoded by it is met in $\bar{s}$ or not.

Note that the lowest bit β indicates whether such bit sequences need to be fetched or not. Encoding this in the individual transition entries is in fact redundant, since for each particular $\Pi[i]$ and label a, either all transitions with that label are independent, or they are not. However, storing this dependency information per transition only costs one bit per entry, while the benefit is that additional memory accesses can be completely avoided for independent transitions.

3.2 Successor generation

At the start of a search iteration, each block fetches a tile of new state vectors from the global memory. How this is done is explained at the end of Sect. 3.3. The tile size depends on the block size *BlockSize*.

On GPUs, one should realise fine-grained parallelism to obtain good speedups. Given the fact that each state vector consists of n states, and the outgoing transitions information of different $\Pi[i]$ needs to be fetched from physically separate parts of the memory, it is reasonable to assign n threads to each state vector to be explored. In other words, in each iteration, the tile size is at most $BlockSize/n$ vectors. Assigning multiple threads per LTS for fetching, as done by Hong et al. [20], does not lead to further speedups, since the number of transition entries to fetch is usually quite small due to the sparsity of the LTSs, as observed before by us [44].

Algorithm 2 presents in pseudo-code our approach to successor generation. At line 1, several arrays are allocated in the (block-local) shared memory. They are declared **extern**, meaning that their size is given when launching the kernel. In practice, all available shared memory is claimed for a single array, which is then divided into the *tile*, B, *cache*, and *cnt* arrays.

We group the threads into vector groups of size n to assign them to state vectors. Each vector group has a block-local unique ID, which we refer to as the VGID (line 3). Since each vector group is assigned to a state vector, we have up to $BlockSize/n$ vector groups per block. For a vector $\bar{s}$, each thread with ID i w.r.t. its vector group (its VGTID, see line 4) fetches the outgoing transitions of $\bar{s}[i+1]$. At line 6, $\bar{s}$ is fetched from the work tile, and at line 7, $\bar{s}[i+1]$ is retrieved. Next, at lines 8–10, the necessary offsets are calculated to fetch the outgoing transitions of $\bar{s}[i+1]$.

The purpose of the B array is to buffer the outgoing transitions fetched by the threads. In particular, for all transitions with $\beta = 1$, cooperation between the threads in a group must be achieved, to identify possible synchronisations. It is important that the size of B is chosen such that all relevant transitions can be buffered, but catering for the worst case, i.e. expecting n times the maximum branching factor, may exceed the amount of available shared memory. Therefore, we opt for fetching the transitions in several iterations. The threads iterate over their transitions in order of label ID (LID). To facilitate this, the outgoing transitions in *TransArray* of each state have been sorted on LID before exploration started.

Algorithm 2 Multi-threaded successor generation

```
    extern volatile _shared_ unsigned int $tile[], B[], cache[], cnt[]$
2:  < fill tile with state vectors to explore > (Alg. 5)
    $VGID \leftarrow ThreadId/n$
4:  $VGTID \leftarrow ThreadId \mod n$
    if $VGID < |tile|$ then
6:    $\bar{s} \leftarrow tile[VGID]$
      $s \leftarrow \bar{s}[VGTID + 1]$
8:    $procoffset \leftarrow ProcOffsets[VGTID]$
      $stateoffset1 \leftarrow StateOffsets[procoffset + s]$
10:   $stateoffset2 \leftarrow StateOffsets[procoffset + s + 1]$
      $startB \leftarrow (VGID \cdot n + VGTID) \cdot M$
12:   $active \leftarrow$ false
      if $VGTID = 0$ then
14:     $cnt[VGID] \leftarrow (VGID < |tile|)\ ?\ 1 : 0$
    while $cnt[VGID] = 1$ do
16:   if $\neg active$ then
        reset $B[startB] \ldots B[startB + M]$
18:     while $stateoffset1 < stateoffset2$ do
          $T \leftarrow TransArray[stateoffset1]$
20:       if $\neg T.\beta$ then
            $\bar{s}' \leftarrow \bar{s}$
22:         for all $Tgt \in T$ do
              $\bar{s}'[VGTID + 1] \leftarrow Tgt$
24:           if $\neg$STOREINCACHE($\bar{s}'$) then
                STOREINGLOBALHASHTABLE($\bar{s}'$)
26:         $stateoffset1 \leftarrow stateoffset1 + 1$
          else
28:         break
        $i \leftarrow startB$
30:     if $stateoffset1 < stateoffset2$ then
          $active \leftarrow$ true
32:       getAct($act1, T$)
          $B[i] \leftarrow T$
34:       $i \leftarrow i + 1$
          $stateoffset1 \leftarrow stateoffset1 + 1$
36:       while $stateoffset1 < stateoffset2$ do
            $T \leftarrow TransArray[stateoffset1]$
38:         getAct($act2, T$)
            if $act1 = act2$ then
40:           $B[i] \leftarrow T$
              $i \leftarrow i + 1$
42:           $stateoffset1 \leftarrow stateoffset1 + 1$
            else
44:           break
      _syncThreads()
46:   if $VGTID = 0$ then
        $cnt[VGID] \leftarrow$ min. of $B[startB].\alpha \ldots B[startB + n \cdot M].\alpha$
48:   _syncThreads()
      if $active$ and $cnt[VGID] = B[startB].\alpha$ then
50:     $active \leftarrow$ false
        for all $\bar{t} \in \mathcal{V}[cnt[VGID]]$ do
52:       if OWNS($VGTID, \bar{t}$) $\wedge$ ISAPPLICABLE($\bar{t}$) then
            for all $\bar{s}' \in$ CONSTRUCTSUCC($\bar{s}, \bar{t}$) do
54:           if $\neg$STOREINCACHE($\bar{s}'$) then
                STOREINGLOBALHASHTABLE($\bar{s}'$)
56:   _syncThreads()
      if $VGTID = 0$ then
58:     $cnt[VGID] \leftarrow 0$
      _syncThreads()
60:   if $stateoffset1 < stateoffset2 \vee active$ then
        $cnt[VGID] \leftarrow 1$
```

The buffer size required for a single thread to store transitions with the same LID can now be determined before exploration as the maximum number of outgoing transitions with the same LID for any state in any of the $\Pi[i]$, where $\beta = 1$. In Algorithm 2, we refer to this size as M. At line 11, the start offset for a thread i in B is calculated using M: It depends on the ID of the group, i.e. $VGID \cdot n$ threads have buffer areas that are physically before the one of i, and the group thread id, since $VGTID$ threads also have buffer areas before the one of i. Finally, this total is multiplied by M to obtain the absolute address of the area of i.

Note that B is not required for transition entries with $\beta = 0$. These can directly be processed, and the corresponding target state vectors can immediately be stored for duplicate detection (see Sect. 3.3).

At line 12, the *active* variable is set to false. The purpose of this variable is to indicate whether a thread still has ‘active’ transition entries in its buffer, i.e. whether the buffer contents still needs to be processed. Successor generation is performed in the while-loop starting at line 15; its condition depends on the value of $cnt[VGID]$, which is set, when checking, to 1 as long as the vector group still has work to be done with the current state vector, and 0 otherwise. Threads with a non-active buffer contents first reset their buffer area at line 17, after which the next outgoing transition is fetched. If the transition does not require synchronisation (line 20), the new successor state vectors are constructed (lines 21–23) and stored in a local cache (line 24). If the cache is full, the thread tries to store new vectors immediately in a global hash table (line 25). In Section 3.3, a detailed explanation of both the local caches and the global hash table is given.

Alternatively, if the newly fetched transition requires synchronisation, subsequent transition entries with the same LID are fetched and stored in the buffer area (lines 30–44). Note that *active* is then set to true to indicate that the buffer contents requires work. Since groups may cross warp boundaries, the threads in a group are not necessarily automatically synchronised. Because of this, a block-local synchronisation is required (line 45), after which the vector group *leader*, the one with $VGTID = 0$, determines the lowest LID currently in the buffer of the vector group and stores this value in $cnt[VGID]$ (lines 46–47).

After another synchronisation, all threads with the lowest LID, i.e. those with transitions labelled $cnt[VGID]$ (line 49) fetch all relevant synchronisation vectors from the global memory (line 51). We say that thread i *owns* synchronisation vector $\bar{t}$ iff there is no $j \in 1..n$ with $j < i$ and $\bar{t}[j] \neq \bullet$. If a synchronisation vector is owned by the current thread, and applicable w.r.t. the current vector group buffer contents (line 52), then it is used to construct all derivable successors (line 53), which are then stored in the local cache, or, if not possible, the global hash table. Finally, at lines 57–61, $cnt[VGID]$ is set to represent whether at least one thread still has work to be done.

To further illustrate the successor generation procedure, Figure 4 provides an example situation for a vector with

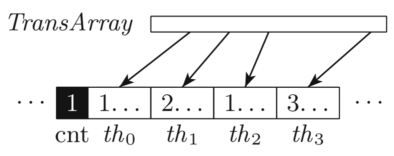

Fig. 4 Fetching transitions

$n = 4$. Threads th_0 to th_3 have fetched transitions with the lowest LIDs for their respective process states that have not yet been processed in the successor generation, and thread th_0 has determined that the next lowest LID to be processed by the vector group is 1. This value is written in the *cnt* location. Since transitions in *TransArray* are sorted per state by LID, we know that all possible transitions with LID $= 1$ have been placed in the vector group buffer. Next, all threads that fetched entries with the lowest LID, in the example threads th_0 and th_2, start scanning the encodings of synchronisation vectors in $\mathcal{V}$ applicable on that LID. If a thread encounters a rule that it owns, then it checks the buffer contents to determine whether the rule is applicable. If it is, it constructs the target state vectors and stores them for duplicate detection. In the next iteration, all entries with lowest LID are removed, the corresponding threads fetch new entries, and the vector group leader determines the next lowest LID to be processed.

3.3 Closed set maintenance

Local state caching As explained in Section 2, we choose to use a global memory hash table to store states. Research has shown that in state space exploration, due to the characteristics of most networks, there is a strong sense of locality, i.e. in each search iteration, the set of new state vectors is relatively small, and most of the already visited vectors have been visited about two iterations earlier [28,36]. This allows effective use of block local *state caches* in shared memory. Such a cache, implemented as a linear probing hash table, can be consulted quickly, and many duplicates can already be detected, reducing the number of global memory accesses. We implemented the caches in a lockless way, apart from using a compare-and-swap (CAS) operation to store the first integer of a state vector.

When processing a tile, threads add successors to the cache. When finished, the block scans the cache, to check the presence of the successors in the global hash table. Thus, caches also allow threads to cooperatively perform global duplicate detection and insertion of new state vectors.

Global hash table For the global hash table, we initially used the Cuckoo hash table of Alcantara et al. [2]. Cuckoo hashing has the nice property that lookups are done in constant time, namely it requires H memory accesses, with H the number of hash functions used.

However, an important aspect of Cuckoo hashing is that elements are relocated in case collisions occur. Algorithm 3 describes how looking up and inserting elements, i.e. a find-or-put operation, for a Cuckoo hash table functions. A new element is inserted using an atomic exchange operation (**atomicExch**); given a memory address and the element to store, the exchange procedure atomically stores the element at the address and returns the old value stored there. If the

Algorithm 3 Find-or-put for Cuckoo hash table

```
    $location \leftarrow h_0(\bar{s})$
2:  for $i = 0$ to $maxiters$ do
      $entry \leftarrow$ atomicExch($\&Visited[location], \bar{s}$)
4:    if $entry =$ empty $\vee$ $entry = \bar{s}$ then
        return true
6:    $\bar{s} \leftarrow entry$
      for $j = 0$ to $H$ do
8:      $location2 \leftarrow h_j(\bar{s})$
        if $location2 = location$ then
10:       $location \leftarrow (location2 + 1) \mod H$
          break
12: return false
```

memory location was initially empty or already contained the given element, then the find-or-put can terminate (line 4–5). Otherwise, the old element needs to be relocated to a new address using the next in line hash function. This relocating of elements is allowed up to a predetermined number of times (*maxiters*).

Alcantara et al. [2] store key-value pairs in 64-bit integers, on which atomic exchange operations can still be applied. In model checking, however, state vectors frequently require more than 64 bits, ruling out atomic insertions. After having created our own extension of the hash table of Alcantara et al. [2] that allows for larger elements, we experienced in experiments that the number of explored states far exceeded the actual number of reachable states, showing that in many cases, threads falsely conclude that a vector was not present (a *false negative*). This effect is already noted by Alcantara et al. [2] for 32 and 64-bit elements; consider an element A being added to the hash table at location l, and after that, an element B is inserted in the same location, causing A to be moved to another location. Now, if a thread looks for A starting at l, it will find B, conclude that A is not present in the hash table, swap A and B, and move B to another location. In the resulting situation, A is stored twice in the hash table. This effect tends to get worse for elements that do not allow atomic insertions. In that case, threads may read elements which have only partially been stored at that moment. When this happens, the involved thread either needs to wait until the storing has finished, which is hard to manage, or move on to another location, missing the opportunity to identify a duplicate. The latter option also has another bad side-effect: we experienced that many times, states could not be placed at any of their H potential locations. In order for the exploration to continue, we had to assume in those cases that the state vector was already present, making the exploration potentially non-exhaustive. In Sect. 5, we report on the effects of using Cuckoo hashing in our GPU exploration tool.

In order to decrease the number of false negatives and guarantee that explorations are exhaustive, we chose as an alternative to implement a hash table using buckets, linear probing and bounded double hashing. It is implemented using an array, each consecutive *WarpSize* 32-bit integers forming a bucket. This plays to the strength of warps: when a block of threads is performing duplicate detection, all the threads in a warp cooperate on checking the presence of a particular $\overline{s}$.

The hash functions we use are constructed in a fashion similar to the one proposed by Alcantara et al. [2]. Each function $h_i : \mathcal{S}_{\mathcal{M}} \rightarrow \mathbb{N}$ is defined as

$$h_i(\overline{s}) = \sum_{j \in 1..c} (a_i \cdot \overline{s}_j + b_i) \bmod P \bmod \mathit{NrOfBuckets}$$

Here, the $\overline{s}_j$ represent the integers that together store $\overline{s}$, c being the number of integers required to store a state vector. Furthermore, P is a predefined large prime number, the total number of buckets in the hash table is represented by *NrOfBuckets*, and each h_i has two randomly generated constants a_i and b_i. For our hash table, we use two hash functions. The first hash function h_0 is used to find the primary bucket, and the second one h_1 is used to jump to another bucket each time a full bucket has been encountered. A warp can fetch a bucket with one memory access, since the bucket size directly corresponds with one cache line. Subsequently, the bucket contents can be checked in parallel by the warp. This is similar to the *walk-the-line* principle of Laarman et al. [22]; instead that here, the walk is done in parallel, so we call it *warp-the-line*. Note that each bucket can contain up to $\mathit{WarpSize}/c$ vectors, with c the number of 32-bit integers required for a vector. Hence, the underlying assumption is that a single state vector never requires more than *WarpSize* integers. If this is not the case, though, one can generalise the technique to each warp fetching multiple consecutive buckets.

If the vector is not present and there is a free location, the vector is inserted. If the bucket is full, h_1 is used to jump to another bucket, and so on. This is similar to the approach of Dietzfelbinger et al. [13]; instead that we do not move elements between buckets.

Algorithm 4 Warped find-or-put for hash table with buckets (single integer state vectors version)

```
    extern volatile _shared_ unsigned int cache []
2:  < process work tile and fill cache with successors > (Alg. 2)
    WarpId ← ThreadId / WarpSize
4:  WarpTId ← ThreadId mod WarpSize
    i ← WarpId
6:  while i < |cache| do
      $\overline{s}$ ← cache[i]
8:    if isNewVector($\overline{s}$) then
        for j = 0 to H do
10:       BucketId ← $h_0(\overline{s})$
          entry ← Visited[BucketId + WarpTId]
12:       if entry = $\overline{s}$ then
            setOldVector(cache[i])
14:       $\overline{s}$ ← cache[i]
          if isNewVector($\overline{s}$) then
16:         for l = 0 to WarpSize do
              if Visited[BucketId + l] = empty then
18:             if WarpTId = 0 then
                  old = atomicCAS(&Visited[BucketId + l], empty, $\overline{s}$)
20:               if old = empty then
                    setOldVector($\overline{s}$)
22:           if ¬isNewVector($\overline{s}$) then
                break
24:       if ¬isNewVector($\overline{s}$) then
            break
26:       BucketId ← BucketId + $h_1(\overline{s})$
      i ← i + BlockSize/WarpSize
```

The pseudo-code for scanning the local cache and performing a find-or-put operation in the case that state vectors fit in a single 32-bit integer is displayed in Algorithm 4. The implementation contains the more general case. Once a work tile has been explored and the successors are in the cache, each thread participates in its warp to iterate over the cache contents (lines 6–27). If a vector is new (line 8, note that empty slots are marked 'old'), insertion in the hash table will be tried up to $H \in \mathbb{N}$ times. At lines 11–13, warp-the-line is performed, each thread in a warp investigating the appropriate bucket slot (the *Visited* array contains the buckets). If any thread sets $\overline{s}$ as old at line 13, then all threads will detect this at line 15 since $\overline{s}$ is read from shared memory. If the vector is not old, then it is attempted to insert the vector in the bucket (lines 15–23). This is done by the warp leader ($\mathit{WarpTId} = 0$, line 18), by performing a CAS. CAS takes three arguments, namely the address where the new value must be written, the expected value at the address, and the new value. It only writes the new value if the expected value is encountered and returns the encountered value; therefore, a successful write has happened if **empty** has been returned (line 20). Finally, in case of a full bucket, h_1 is used to jump to the next one (line 26).

We experienced good speedups and no unresolved collisions using a double hashing bound H of 8, and, although still present, far fewer false negatives compared to Cuckoo hashing. For a detailed discussion of this, see Sect. 5. Finally, it should be noted that chaining is not a suitable option on a GPU since it requires memory allocation at runtime, and the required sizes of the chains are not known a priori.

Recall that the two important data structures are *Open* and *Visited*. Given the limited amount of global memory, and that the state space size is unknown a priori, we prefer to initially allocate as much memory as possible for *Visited*. But also the required size of *Open* is not known in advance, so how much memory should be allocated for it without potentially wasting some? We choose to combine the two in a single hash table using the highest bit in each vector encoding to indicate whether it should still be explored or not. The drawback is that unexplored vectors are not physically close to each other in memory, but the typically large number of threads can together scan the memory relatively fast, and using one data structure drastically simplifies implementation. It has the added benefit that load-balancing is handled by the hash functions, due to the fact that the distribution over the hash table achieves distribution over the workers. A consequence is that the search will not be strictly BFS, but this is not a requirement.

Algorithm 5 Work scanning

```
    extern volatile _shared_ unsigned int $tile[]$, $tilecount$
2:  $WarpId \leftarrow ThreadId\ /\ WarpSize$
    $WarpTId \leftarrow ThreadId \mod WarpSize$
4:  $GlobalWarpId \leftarrow (NrOfBlocks/WarpSize) \cdot BlockId + WarpId$
    $NrOfWarps \leftarrow (BlockSize/WarpSize) \cdot NrOfBlocks$
6:  $i \leftarrow GlobalWarpId$
    while $i < NrOfBuckets \wedge tilecount < (BlockSize/n)$ do
8:    $\bar{s} \leftarrow Visited[(i \cdot WarpSize) + WarpTId]$
      if ISNEWVECTOR($\bar{s}$) then
10:     $j \leftarrow$ atomicAdd($\&tilecount$, 1)
        if $j < BlockSize/n$ then
12:       $tile[j] \leftarrow \bar{s}$
          SETOLDVECTOR($Visited[(i \cdot WarpSize) + WarpTId]$)
14:   $i \leftarrow i + NrOfWarps$
```

The pseudo-code for scanning the global hash table in order to gather a tile of work is presented in Algorithm 5, again for the case that state vectors fit in a single 32-bit integer. The warps iterate over the buckets in the global hash table, hence each thread must be aware of the global ID of the warp it resides in, the ID of itself w.r.t. that warp, and the total number of warps. As long as the tile is not filled and there are buckets left to scan (line 7), the scanning continues. Variable *tilecount* indicates the current tile size. At line 8, a state vector is fetched from the bucket assigned to the current warp, and if this is a new state (line 9), *tilecount* is atomically increased using the **atomicAdd** operation. In case the new tile size does not exceed the maximum size, the state vector is added to the tile, and the original state in the global hash table is marked old.

As mentioned, work scanning can be done reasonably fast, but in practice it still represents a performance bottleneck. In the next section, we discuss possible extensions to improve performance.

3.4 Further extensions

On top of the basic approach, we implemented the following extensions. First of all, instead of just one, we allow a variable number of search iterations to be performed within one kernel launch. This improves duplicate detection using the caches due to them maintaining more of the search history (shared memory data are lost once a kernel terminates).

Second of all, building on the first extension, we implemented a technique we call *work claiming*. When multiple iterations are performed per launch, and a block is not in its final iteration, its threads will immediately add the unexplored successors they generated in the current iteration to their own work tile for the next iteration. In other words, while adding new state vectors to the global hash table, each block claims those vectors immediately for exploration work in the next iteration. This reduces the need for scanning for new work at the start of that iteration. In the final iteration of a kernel launch, though, this cannot be done since the contents of shared memory is wiped whenever a kernel run terminates.

Work claiming does not cause too much imbalance in the work loads of the blocks. Blocks that do not produce many new state vectors may scan for work at the start of the next iteration, while blocks that do produce many vectors will only claim a fixed number of them (depending on the maximum tile size), leaving the remaining vectors to be gathered by other blocks.

4 Checking for deadlocks and safety property violations

The many-core exploration technique we described in Sect. 3 can be straightforwardly extended to support verifying functional properties that can be checked through reachability analysis. In particular, deadlock detection and the detection of violations of safety properties can be added.

Deadlock detection involves keeping track of whether from each reachable state, at least one transition is enabled. This is not completely trivial, since in the many-core exploration technique, multiple threads collaborate on checking the outgoing transitions of a state vector. In other words, the threads in a vector group need to communicate with each other whether they were able to reach at least one successor or not. We enabled this communication by reusing the B array between successor generation iterations (Algorithm 2). Whenever a deadlock state is detected, a global flag is set to indicate that the exploration should immediately terminate.

Safety properties require a more involved mechanism, but they can straightforwardly be added to the input by extending the networks of LTSs with an LTS representing the property. This property LTS is constructed in such a way that whenever the system can perform an action which is relevant for the property, then the property LTS will synchronise with the system on that action, thereby changing state as well. Then, during the exploration, it must be checked whether for each constructed successor state vector, the property LTS is not in a predefined error state. When it is, the exploration can immediately terminate. Extending an LTS with a predefined error state s_p^F actually results in a finite automaton for which s_p^F is a final state.

It should be noted that property LTSs actually correspond with so-called *monitors* in runtime verification, i.e. they can be used to monitor a property while exploring the state space. As explained in [8], the class of monitorable languages is actually more expressive than the class of safety properties, meaning that it is more precise to claim that our exploration technique supports checking monitorable properties.

To facilitate the design, we use for the properties finite automata (property LTSs) defined over an alphabet of regular expressions. The regular expressions correspond to sets of actions from the component LTSs of the model, i.e. LTS vector Π. More formally, the property automaton A_p is a tuple

$(\mathcal{S}_p, \mathcal{A}_p, \mathcal{T}_p, s_p^0, s_p^F)$, where $\mathcal{S}_p$ is a finite set of states, $\mathcal{A}_p$ is a finite set of actions, $\mathcal{T}_p \subseteq S_p \times A_p \times S_p$ is a finite set of transitions, $s_p^0 \in S_p$ is the initial state, and $s_p^F \in S_p$ is a final state. Intuitively, since we work with the negation of the verified property, the final state s_p^F is an error state. Let ρ be a function from $\mathcal{A}_p$ to $2^{\cup_i \mathcal{A}_i}$, which assigns to each action-regular expression $r \in \mathcal{A}_p$ a set of actions from Π satisfying r. The regular automaton is translated to a property automaton $A'_p = (\mathcal{S}_p, \mathcal{A}'_p, \mathcal{T}'_p, s_p^0, s_p^F)$, where $\mathcal{A}'_p = \{a \mid \exists r \in \mathcal{A}_p . a \in \rho(r)\}$, and $(s, a, s') \in \mathcal{T}'_p$ iff $(s, r, s') \in \mathcal{T}_p$ and $a \in \rho(r)$.

A network of LTSs $\mathcal{M}$ and an automaton (property LTS) A'_p can be combined into a network $\mathcal{M}' = \langle \Pi', \mathcal{V}' \rangle$, in which

- $\Pi' = \langle \Pi[1], \ldots, \Pi[n], A'_p \rangle$.
- $\mathcal{V}' = \{\langle \bar{t} \oplus \langle \bullet \rangle, a \rangle \mid \langle \bar{t}, a \rangle \in \mathcal{V} \wedge a \notin \mathcal{A}'_p\} \cup \{\langle \bar{t} \oplus \langle a \rangle, a \rangle \mid \langle \bar{t}, a \rangle \in \mathcal{V} \wedge a \in \mathcal{A}'_p\}$.

In the above formula, $\oplus$ is a concatenation operator for vectors. $\mathcal{V}'$ is constructed in such a way, that all transitions in the property LTS must synchronise with transitions with the same label that are enabled in the system. Now, whenever during the search a state vector is generated that contains the final state of A'_p, a property violation can be signalled.

An example of a property automaton A_p that corresponds to a mutual exclusion property for a network of two process LTSs is given in Fig. 5. The intuition is that from the initial state s_p^0 with action *CSin1* (resp. *CSout2*) LTS 1 (resp. 2) enters the critical section state s_p^1 (resp. s_p^2), whereas any *CSout* action leads to the error (final) state s_p^F since a process that is not in the critical section is not able to leave it. The *CSout* actions of both LTSs are captured by the regular expression *"CSout[1|2]"*. From the critical section state s_p^1 (resp. s_p^2) we can go back to the initial state s_p^0 with action *CSout1* (resp. *CSout2*). Any other action, represented by the regular expression *"(CSout2|CSin[1|2])"* (resp. *"(CSout1|CSin[1|2])"*) results in the error state s_p^F. The occurrence of a *CSin2* (resp. *CSin1*) action indicates the erroneous situation that the two processes are both in the critical section, while the other actions should be disabled since they do not correspond with the current situation (only processes in the critical section can leave it, and only processes outside the critical section can enter it).

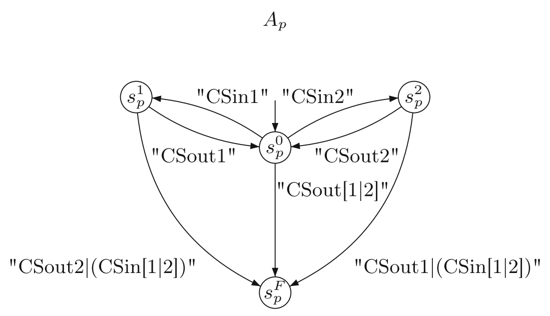

Fig. 5 Automaton A_p over regular expressions for a mutual exclusion property

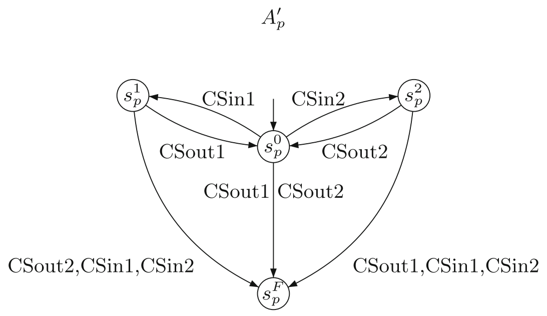

Fig. 6 Property automaton (LTS) A'_p that corresponds to automaton A_p

Automaton A'_p corresponding to A_p is given in Fig. 6. The regular expressions of A_p are unfolded to the corresponding actions of A'_p. For instance, unfolding *"(CSout1|CSin[1|2])"* results in the three separate actions *CSout1*, *CSin1*, and *CSin2*.

5 Implementation and experiments

We implemented the exploration techniques in CUDA C using the CUDA toolkit 5.5.[2] The implementation was tested using 25 models from different sources; some originate from the distributions of the state-of-the-art model checking toolsets CADP [15] and mCRL2 [11], and some from the BEEM database [35]. In addition, we added two we created ourselves. Here, we discuss the results for a representative subset.

Sequential experiments have been performed using EXP.OPEN [25] with GENERATOR, both part of CADP. These are highly optimised for sequential use. Those experiments were performed on a machine with an INTEL XEON E5520 2.27 GHz CPU, 1 TB RAM, running Fedora 12. The GPU experiments were done on machines running CentOS Linux, with a Kepler K20m GPU, an INTEL E5- 2620 2.0 GHz CPU, and 64 GB RAM. The GPU has 13 SMs, 5 GB global memory (realising a hash table with about 1.3 billion integers), and 48kB (12,288 integers) shared memory per block. We chose not to compare with the GPU tool of [14], since it is a CPU-GPU hybrid and, therefore, does not clearly allow to study to what extent a GPU can be used by itself for exploration. Furthermore, it uses bitstate hashing, thereby not guaranteeing exhaustiveness.

[2] The implementation and experimental data is available at http://www.win.tue.nl/~awijs/GPUexplore. For the CUDA toolkit, see http://developer.nvidia.com/cuda-zone.

We also decided not to experimentally compare our techniques with the ones proposed by Bartocci et al. [7], for the same reasons given by them for not comparing with our work. Too many aspects are different in design to allow for a fair comparison. The required inputs of the tools are very different and require quite some manual preparation at the moment. Even when inputs are obtained that fundamentally describe the same system, the derived state spaces are usually very different in terms of number of states and transitions. This directly relates to the underlying frameworks: performance comparisons between SPIN and action-based model checkers such as the mCRL2 and CADP toolsets are very hard to fairly perform for the same reason.

Some experiments were conducted with the model checker LTSMIN [23] using the six CPU cores of the machines equipped with K20s. LTSMIN uses the most scalable multicore exploration techniques currently available.

Table 1 displays the characteristics of the models we consider here. The first five are models taken from and inspired by those distributed with the mCRL2 toolset (in general '.1' suffixed models indicate that we extended the existing models with extra independent actions to obtain larger state spaces). The non-extended models have the following origins:

Table 1 Benchmark characteristics

Model	#States	#Transitions	#Bits $\overline{s}$
1394	198,692	355,338	36
1394.1	36,855,184	96,553,318	49
ACS	4,764	14,760	35
ACS.1	200,317	895,004	41
WAFER STEPPER.1	4,232,299	19,028,708	32
ABP	235,754,220	945,684,122	78
BROADCAST	60,466,176	705,438,720	70
TRANSIT	3,763,192	39,925,524	37
CFS.1	252,101,742	1,367,483,201	83
ASYN3	15,688,570	86,458,183	65
ASYN3.1	190,208,728	876,008,628	70
ODP	91,394	641,226	31
ODP.1	7,699,456	31,091,554	40
DES	64,498,297	518,438,860	49
LAMPORT.8	62,669,317	304,202,665	36
LANN.6	144,151,629	648,779,852	37
LANN.7	160,025,986	944,322,648	45
PETERSON.7	142,471,098	626,952,200	62
SZYMANSKI.5	79,518,740	922,428,824	45

- The 1394 model is written in the mCRL2 modelling language and describes the 1394 or firewire protocol. It has been created by Luttik [27].
- The ACS model describes a part of the software of the ALMA project of the European Southern Observatory, which involves controlling a large collection of radio telescopes. It consists of a manager and some containers and components. The model was created by Ploeger [38].
- WAFER STEPPER.1 is an extension of a model of a wafer stepper included in the mCRL2 toolset.

The next two have been created by us. These can be described as follows:

- ABP is a model consisting of six independent subsystems, each involving two processes communicating using the Alternating Bit Protocol.
- BROADCAST consists of ten independent subsystems, each containing three processes that together achieve a barrier synchronisation via a sequence of two-party synchronisations.

The seven models after that originate from CADP. Their origins are the following:

- The TRANSIT model describes a Transit-Node [31].
- CFS.1 is an extended model of a coherency protocol of the Cluster File System [34].
- ASYN3 describes the asynchronous Leader Election protocol used in the HAVi (Home Audio-Video) standard, involving three device control managers. The model is fully described by Romijn [39].
- ODP is a model of an open distributed processing trader [17].
- The DES model describes an implementation of the data encryption standard, which allows to cipher and decipher 64-bit vectors using a 64-bit key vector [32].

The final five models were taken from the BEEM database. These models have first been translated manually to mCRL2 since our input, network of LTSs, uses an action-based representation of system behaviour, but BEEM models are state-based and hence this gap needed to be bridged. In the following, we briefly describe these models:

- LAMPORT.8 is an instance of Lamport's Fast Mutual Exclusion algorithm [24].
- LANN.6 and LANN.7 are two instances of the Lann Leader Election algorithm for Token Rings [16].
- PETERSON.7 is an instance of Peterson's Mutual Exclusion protocol [37].
- SZYMANSKI.5 is an instance of Szymanski's Mutual Exclusion protocol [40].

The characteristics given in Table 1 for each model are the total number of states reachable from the initial state, the

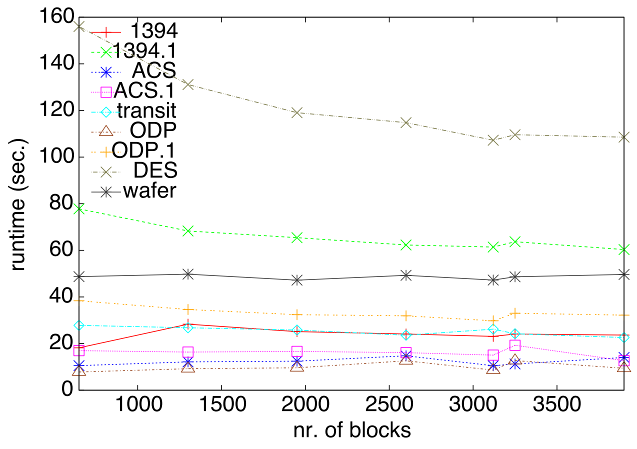

Fig. 7 Performance with varying nr. of blocks I (iters = 10)

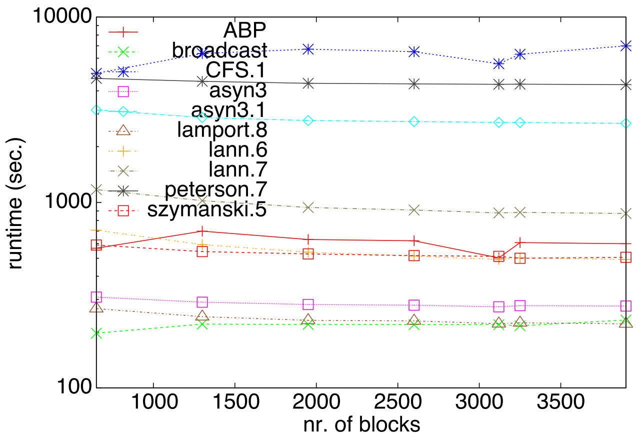

Fig. 8 Performance with varying nr. of blocks II (iters = 10)

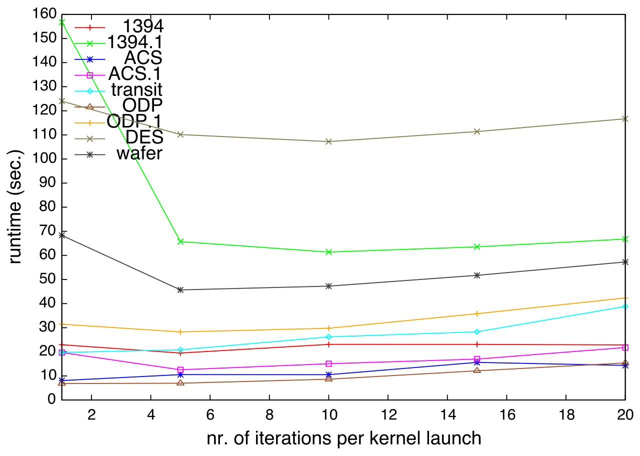

Fig. 9 Performance with varying nr. of iterations per kernel I (blocks = 3120)

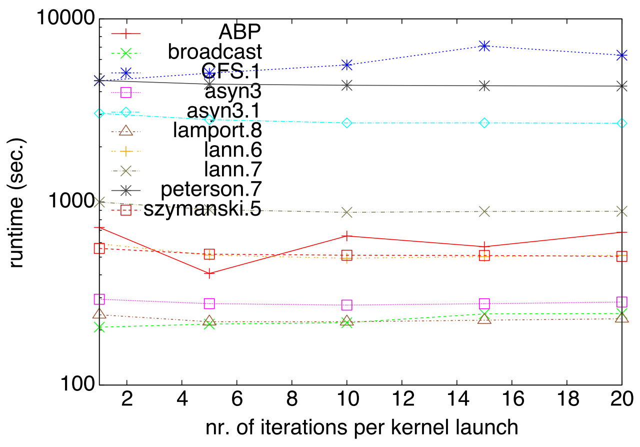

Fig. 10 Performance with varying nr. of iterations per kernel (blocks = 3120)

number of reachable transitions, and the size in bits of each state vector.

An important question is how the exploration should be configured, i.e. how many blocks should be launched, and how many iterations should be done per kernel launch. Regarding the block size, we ended up selecting 512 threads per block, since other numbers of threads resulted in reduced performance. We tested different configurations with that block size, using double hashing with work claiming; Figs. 7 and 8 show our results launching a varying number of blocks (note the logscale in Fig. 8), each performing 10 iterations per kernel launch. The ideal number of blocks for the K20m seems to be 240 per SM, i.e. 3120 blocks. For GPU standards, this is small, but launching more often negatively affects performance, probably due to the heavy use of shared memory.

Figures 9 and 10 show our results on varying the number of iterations per kernel launch. Here, it is less clear which value leads to the best results, either 5 or 10 seems to be the best choice. With a lower number, the more frequent hash table scanning becomes noticable, while with higher numbers, the less frequent passing along of work from SMs to each other leads to too much redundancy, i.e. re-exploration of states, causing the exploration to take more time.

For further experimentation, we opted for 10 iterations per launch. Figures 11 and 12 show our runtime results (note the log scale). The GPU extension combinations used are Double Hashing (DH), DH+work Claiming (DH+C), and DH without local caches (NC). The smaller state spaces are represented in Fig. 11. Here, DH and NC often do not yet help to speed up exploration; the overhead involved can lead to longer runtimes compared to sequential runs. However, DH+C is more often than not faster than sequential exploration. The small differences between DH and NC and the big ones between NC and DH+C (which is also the case in Fig. 12) indicate that the major contribution of the caches is work claiming, as opposed to localised duplicate detection, which was the original motivation for using the caches. DH+C speeds up DH on average by 42 %.

Also, note that in the smaller cases in Fig. 11, NC tends to outperform DH. Apparently, the effects of localised dupli-

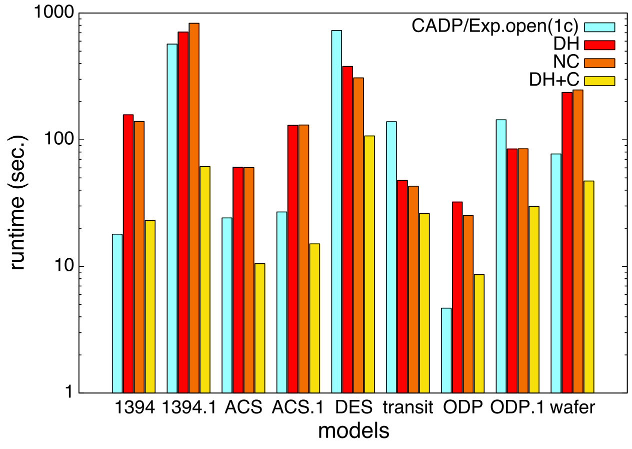

Fig. 11 Runtime results for various tools I

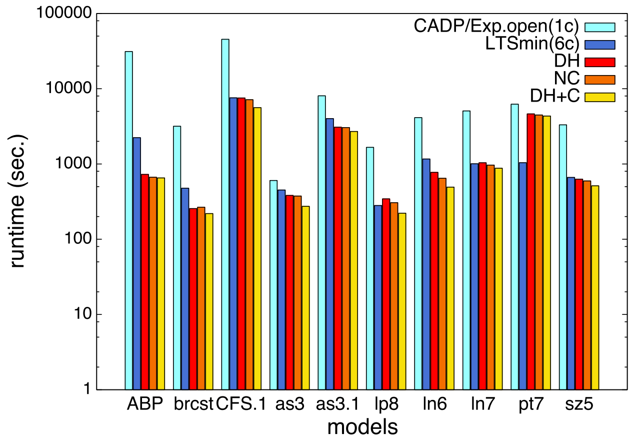

Fig. 12 Runtime results for various tools II

cate detection and more coalesced global hash table accesses with fewer parallelism only have a positive impact when the state space has a considerable size. In NC, local caches are absent, so each thread needs to check for duplicates independently, i.e. for complete state vectors, whereas in DH, this is done by all threads in a warp in collaboration. The BROADCAST and ABP cases show impressive speedups of almost two orders of magnitude. Both involve large state vectors involving 30 processes, showing that successor generation is particularly efficient if many threads can collaborate on exploring the outgoing transitions of a given state vector, that is, if there is much potential for fine-grained parallelism. This is a very positive result since industrial-sized models tend to have many processes.

It should be noted that for vectors requiring multiple integers, GPU exploration tends to perform on average 2 % redundant work, i.e. relatively few states are re-explored. In those cases, data races occur between threads writing and reading vectors since only the first integer of a vector is written with a CAS. However, we consider these races benign since it is important that all states are explored, not how many times, and adding additional locks hurts the performance.

Table 2 Work redundancy results with Cuckoo hashing. o.o.t. = out of time (>3 hrs.)

Model	#States	% Redundancy
1394	188,434	−5.2
1394.1	37,760,386	2.4
ACS	3,484	−26.9
ACS.1	149,040	−25.6
WAFER STEPPER.1	3,773,815	−10.8
ABP	o.o.t.	–
BROADCAST	o.o.t.	–
TRANSIT	o.o.t.	–
CFS.1	o.o.t.	–
ASYN3	o.o.t.	–
ASYN3.1	o.o.t.	–
ODP	91,400	0.0
ODP.1	8,395,804	9.0
DES	o.o.t.	–
LAMPORT.8	64,459,411	2.86
LANN.6	o.o.t.	–
LANN.7	o.o.t.	–
PETERSON.7	59,557,610	−58.2
SZYMANSKI.5	o.o.t.	–

To put the average amount of duplicate work performed using our hash table into perspective, we also conducted some experiments with a version of our tool where the hash table with buckets was replaced by the Cuckoo hash table of Alcantara et al. [2]. The results are listed in Table 2. First of all, for many models, the predetermined upper bound for the runtime, set at 3 hours, was reached. This is caused by the fact that we were forced to use a heuristical duplicate detection procedure, already mentioned in Sect. 3.3. After extending the standard Cuckoo hash table find-or-put of Algorithm 3 to support multi-integer entries, we quickly observed that for practically all the state spaces only very small fragments were explored before an error was reported that a state vector could not be added to the hash table. When looking up and storing elements is no longer atomic, threads may encounter vectors in their target locations which have only partially been written at that moment. Then, it must be decided what should happen next, and the only workable option is to try to insert the new vector in another location. However, with state spaces, which tend to have a high locality in terms of revisiting states, we observed that such situations occur very frequently, and in those cases, some new state vectors could not be stored in any of their H possible locations, which causes a hash table error. One can mitigate this by setting H to a higher value, i.e. using more hash functions (the results in Table 2 are for experiments with $H = 7$), but this does not effectively remove most of the erroneous situations.

Therefore, on top of using more hash functions, we had to resort to use the heuristics that if a new vector could not be stored in any of its H possible locations, then it must already have been visited before. This resolves the error situations, but in a number of experiments, it could be observed that the number of new state vectors in the hash table now fluctuated indefinitely; when an old vector $\overline{s}$ is evicted from its location, and a hash table error occurs trying to place $\overline{s}$ somewhere else, it may be effectively removed from the hash table, in case no more copies of it are stored elsewhere. In that case, whenever $\overline{s}$ is revisited again later, it is concluded that it has not yet been explored, and it is again added as a new vector. This causes the total number of new state vectors to increase again. Later, $\overline{s}$ can again be evicted, and this whole scenario may be repeated indefinitely.

Another bad effect of the used heuristics is that explorations are no longer guaranteed to be exhaustive. In fact, in five cases we observed that a significant part of the state space, in one case consisting of even more than half the space, was ignored. This is represented by a negative redundancy percentage in the third column of Table 2. Finally, in the remaining cases, we observed work redundancy ranging between 2.4 and 9.0 %.

Figure 12 also includes results for LTSMIN using six CPU cores. This shows that, apart from some exceptions, our GPU implementation on average has a performance similar to using about 10 cores with LTSMIN, based on the fact that LTSMIN demonstrates near-linear speedups when the number of cores is increased. In case of the exceptions, such as the ABP case, about two orders of magnitude speedup is achieved. This may seem disappointing, considering that GPUs have an enormous computation potential. However, on-the-fly exploration is not a straightforward task for a GPU, and a one order of magnitude speedup seems reasonable. Still, we believe these results are very promising and merit further study. Existing multi-core exploration techniques, such as in [23], scale well with the number of cores. Unfortunately, we cannot test whether this holds for our GPU exploration, apart from varying the number of blocks; the number of SMs cannot be varied, and any number beyond 15 on a GPU is not yet available.

Finally, we have also conducted some experiments involving on-the-fly detection of deadlocks and violations to safety properties. The results of these experiments are displayed in Figs. 13 and 14. For deadlock detection, we used some slightly altered versions of the (originally deadlock-free) models, in which a few deadlocks were introduced. The safety properties that were checked for each model are described in Table 3, together with the outcome of each check (in the '?' column), and the state vector size (in bits) of each model-property combination. The entry 'limited action occurrence' in the property description refers to a property stating that at most two occurrences of a given action (type) a are allowed between two consecutive occurrences of another action (type) b. The 'mandatory precedence' property states that action (type) a is always preceded by action (type) b. 'Bounded response' refers to a property stating that after an occurrence of action a, an action b of a given set must

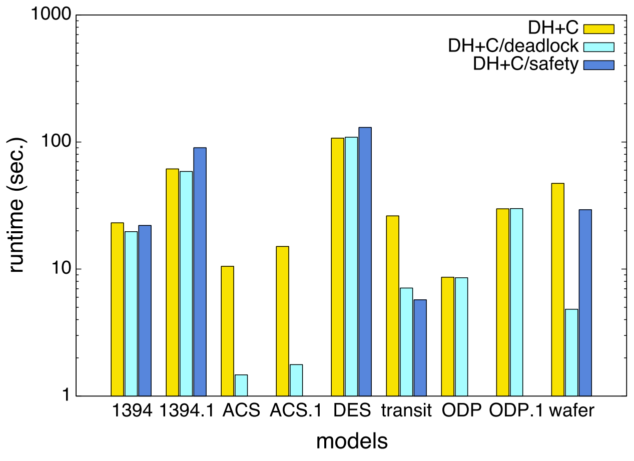

Fig. 13 Runtime results for property checking I

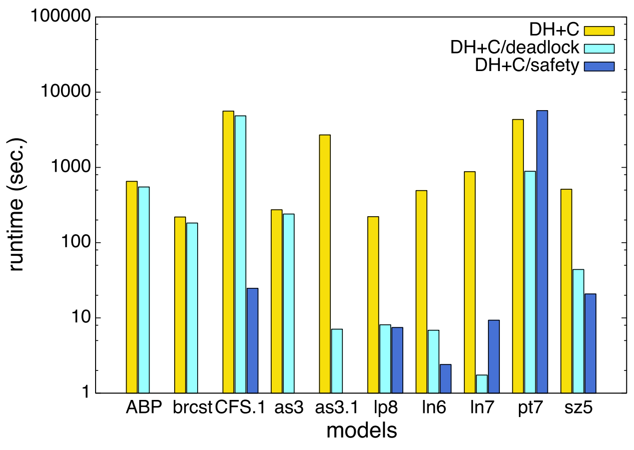

Fig. 14 Runtime results for property checking II

Table 3 Checked safety properties

Model	Property description	?	#Bits $\overline{s}$
1394	Limited action occurrence	✓	39
1394.1	Limited action occurrence	✓	52
WAFER STEPPER.1	Mandatory precedence	✓	34
TRANSIT	Bounded response	✗	39
CFS.1	Limited action exclusion	✗	85
DES	Exact occurrence number	✓	54
LAMPORT.8	Mutual exclusion	✗	39
LANN.6	Mutual exclusion	✗	40
LANN.7	Mutual exclusion	✗	48
PETERSON.7	Mutual exclusion	✓	65
SZYMANSKI.5	Mutual exclusion	✗	48

occur. ‘Limited action exclusion’ is a property in which an action *a* cannot occur between two consecutive occurrences of actions *b* and *c*. In ‘exact occurrence number’, it is required that action *a* occurs an exact number of times, if action *b* has previously occurred. Finally, ‘mutual exclusion’ has the standard meaning (see the example in Sect. 4).

Of course, the runtimes in Figs. 13 and 14 very much depend on where the violations are located in the state spaces, and in what order the states are explored, but the results also provide some interesting insights. In particular, while in most cases, violations could be found significantly faster than the time required to fully explore the state space, in some cases, safety property checking takes more time. The cause of this is that sometimes, by involving the property in the input network of LTSs, the related state space grows in size. The purpose of the property LTS is merely to monitor how the specified system relates to the property in each state, but doing so may cause states which were previously indistinguishable to now be different in terms of the property. On the other hand, any network of LTSs can be checked for deadlocks without alterations, so this effect cannot be observed in any of the deadlock detection experiments.

6 Conclusions

We presented an implementation of on-the-fly GPU state space exploration, proposed a novel GPU hash table, and experimentally compared different configurations and combinations of extensions. Compared to state-of-the-art sequential implementations, we measured speedups of one to two orders of magnitude.

Our choices regarding data encoding and successor generation seem to be effective, and our findings regarding a new GPU hash table, local caches, and work claiming can be useful for anyone interested in GPU graph exploration. Work claiming seems to be an essential mechanism to obtain an efficient, competitive GPU exploration approach.

We also demonstrated that this approach can be extended for efficient on-the-fly checking of deadlocks and violations to safety properties.

We think that GPUs are a viable option for state space exploration. Of course, more work needs to be done to really use GPUs to do model checking. For future work, we will investigate how to support LTS networks that explicitly use data variables and experiment with partial searches [41,42].

Acknowledgments We thank Jan Friso Groote for the help he provided to identify relevant safety properties for the mCRL2 models.

References

1. Alcantara, D.A., Sharf, A., Abbasinejad, F., Sengupta, S., Mitzenmacher, M., Owens, J.D., Amenta, N.: Real-time Parallel Hashing on the GPU. ACM Trans. Graph. **28**(5), 154 (2009)
2. Alcantara, D.A., Volkov, V., Sengupta, S., Mitzenmacher, M., Owens, J.D., Amenta, N.: Building an Efficient Hash Table on the GPU. In GPU Computing Gems Jade Edition. Morgan Kaufmann (2011)
3. Baier, C., Katoen, J.-P.: Principles of Model Checking. The MIT Press, (2008)
4. Barnat, J., Bauch, P., Brim, L., Češka, M.: Designing Fast LTL Model Checking Algorithms for Many-Core GPUs. J. Par. Distr. Comput. **72**, 1083–1097 (2012)
5. Barnat, Jiri, Bauch, Petr, Brim, Lubos, Ceska, Milan: Computing Strongly Connected Components in Parallel on CUDA. In IPDPS, 544–555, (2011)
6. Barnat, Jiri, Brim, Lubos, Ceska, Milan, Lamr, Tomas: CUDA Accelerated LTL Model Checking. In ICPADS, 34–41, (2009)
7. Bartocci, E., DeFrancisco, R., Smolka, S.A.: Towards a GPGPU-Parallel SPIN Model Checker. In SPIN’14. ACM, , 87–96, (2014)
8. Bauer, A.: Monitorability of ω-Regular Languages. CoRR abs/1006.3638, (2010)
9. Bošnački, D., Edelkamp, S., Sulewski, D., Wijs, A.J.: GPU-PRISM: An Extension of PRISM for General Purpose Graphics Processing Units. In joint HiBi / PDMC Workshop (HiBi/PDMC’10), pp. 17–19. IEEE, (2010)
10. Bošnački, D., Edelkamp, S., Sulewski, D., Wijs, A.J.: Parallel Probabilistic Model Checking on General Purpose Graphics Processors. STTT **13**(1), 21–35 (2011)
11. Cranen, S., Groote, J.F., Keiren, J.J., Stappers, F.P., de Vink, E.P., Wesselink, W., Willemse, T.: An overview of the mCRL2 toolset and its recent advances. In TACAS’13, vol. 7795 of LNCS, pp. 199–213. Springer, (2013)
12. Deng, Y.S., Wang, B.D., Shuai, M.: Taming irregular EDA applications on GPUs. In ICCAD’09, pp. 539–546, (2009)
13. Dietzfelbinger, M., Mitzenmacher, M., Rink, M.: Cuckoo hashing with pages. In ESA’11, vol. 6942 of LNCS, pp. 615–627. Springer, (2011)
14. Edelkamp, S., Sulewski, D.: Efficient Explicit-State Model Checking on general purpose graphics processors. In SPIN’10, vol. 6349 of LNCS, pp. 106–123, Springer, (2010)
15. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2010: A toolbox for the construction and analysis of distributed processes. In TACAS’11, vol. 6605 of LNCS, pp. 372–387. Springer (2011)
16. Garavel, H., Mounier, L.: Specification and verification of various distributed leader election algorithms for unidirectional ring networks. Sci. Comp. Program. **29**(1–2), 171–197 (1997)
17. Garavel, H., Sighireanu, M.: A Graphical Parallel Composition Operator for Process Algebras. In FORTE/PSTV’99, vol. 156 of IFIP Conference Proceedings. pp. 185–202. Kluwer (1999)
18. Harish, P., Narayanan, P.J.: Accelerating large graph algorithms on the GPU using CUDA. In HiPC’07, vol. 4873 of LNCS, pp. 197–208. Springer, (2007)
19. Holzmann, G.: Parallelizing the SPIN model checker. In SPIN’12, vol 7385 of LNCS, pp. 155–171. Springer (2012)
20. Hong, S., Kim, S.K., Oguntebi, T., Olukotun, K.: Accelerating CUDA graph algorithms at maximum warp. In PPoPP’11, pp. 267–276. ACM (2011)

21. International Organization of Standardization, Information Processing Systems, Open Systems Interconnection. ISO/IEC. LOTOS. A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807 (1989)
22. Laarman, A., van de Pol, J.C., Weber, M.: Boosting multi-core reachability performance with shared hash tables. In FMCAD'10, pp. 247–255 (2010)
23. Laarman, A., van de Pol, J.C., Weber, M.: Multi-Core LTSmin: Marrying modularity and scalability. In NFM'11, vol. 6617 of LNCS, pp. 506–511. Springer (2011)
24. Lamport, L.: A fast mutual exclusion algorithm. ACM Trans. Comp. Syst. **5**(1), 1–11 (1987)
25. Lang, F.: Exp. open 2.0: a flexible tool integrating partial order, compositional, and on-the-fly verification methods. In IFM'05, vol. 3771 of LNCS, pp. 70–88. Springer (2005)
26. Luo, L., Wong, M., Hwu, W.-M.: An effective GPU implementation of breadth-first search. In DAC'10, pp. 52–55. IEEE Computer Society Press (2010)
27. Luttik, S.P.: Description and formal specification of the link layer of P1394. Technical Report SEN-R9706, CWI (1997)
28. Mateescu, R., Wijs, A.J.: Hierarchical adaptive state space caching based on level sampling. In TACAS'09, vol. 5505 of LNCS, pp. 215–229. Springer (2009)
29. Merrill, D., Garland, M., Grimshaw, A.S.: Scalable GPU graph traversal. In PPoPP'12, pp. 117–128. ACM (2012)
30. Merrill, D., Grimshaw, A.S.: High performance and scalable radix sorting: a case study of implementing dynamic parallelism for GPU computing. Parallel Proc. Lett. **21**(2), 245–272 (2011)
31. Mounier, L.: A LOTOS specification of a transit-node. Spectre Report, pp. 94–8. Verimag, (1994)
32. National Institute of Standards and Technology: Data Encryption Standard (DES). Federal Information Processing Standards, pp. 46–3, (1999)
33. Pagh, R., Rodler, F.F.: Cuckoo Hashing. In ESA'01, volume 2161 of LNCS, pp. 121–133. Springer (2001)
34. Pecheur, C.: Advanced modelling and verification techniques applied to a cluster file system. In ASE'99, pp. 119–126. IEEE (1999)
35. Pelánek, R.: BEEM: benchmarks for explicit model checkers. In SPIN'07, vol. 4595 of LNCS, pp.263–267. Springer (2007)
36. Pelánek, R.: Properties of state spaces and their applications. STTT **10**(5), 443–454 (2008)
37. Peterson, G.L.: Myths about the mutual exclusion problem. Inform. Proc. Lett. **12**(3), 115–116 (1981)
38. Ploeger, B.: Analysis of ACS Using mCRL2. Technical Report 09–11, Eindhoven University of Technology, (2009)
39. Romijn, J.: Model Checking a HAVi Leader Election Protocol. Technical Report SEN-R9915, CWI, (1999)
40. Szymanski, B.K.: Mutual Exclusion Revisited. In JCIT'90, pp. 110–117. IEEE (1990)
41. Torabi Dashti, M., Wijs, A.J.: Pruning state spaces with extended beam search. In ATVA'07, vol. 4762 of LNCS, pp. 543–442. Springer (2007)
42. Wijs, A.J.: What to do next? Analysing and optimising system behaviour in time. PhD thesis, VU University Amsterdam, (2007)
43. Wijs, A.J.: GPU Accelerated strong and Branching Bisimilarity Checking. In TACAS'15, vol. 9035 of LNCS, pp. 368–383. Springer (2015)
44. Wijs, A.J., Bošnački, D.: Improving GPU Sparse Matrix-Vector Multiplication for Probabilistic Model Checking. In SPIN'12, vol. 7385 of LNCS, pp. 98–116. Springer (2012)
45. Wijs, A.J., Bošnački, D.: GPUexplore: Many-Core On-The-Fly State Space Exploration. In TACAS'14, vol. 8413 of LNCS, pp. 233–247. Springer (2014)
46. Wijs, A.J., Katoen, J.-P., Bošnački, D.: GPU-based sraph decomposition into strongly connected and maximal end components. In CAV'14, vol. 8559 of LNCS, pp. 309–325. Springer (2014)